TP 1 : découverte de CAML et de l'environnement de travail pour les TP

I Environnement de travail

I.1 Lancement de la machine virtuelle Ubuntu sous Windows

Une machine virtuelle (VM) est installée avec notre environnement de travail. Pour y accéder, il faut suivre la démarche suivante :

- connexion sous Windows avec les identifiants habituels ;
- accès au dossier « Applications » sur le bureau ;
- accès au sous-dossier « 8-PREPA INFO »
- lancement du raccourci « 1-Ajout_Info_Lvh » (une console apparaît très brièvement et se ferme) ;
- lancement du raccourci « 2-Demarrage_Info_Lvh ».
- Une fois la VM lancée, choisir la session « utilisateur ». Le mot de passe de cette session est « info-lvh » (facile à retenir, c'est le nom de la VM).

I.2 Éditeur de texte et interpréteur OCAML

Pour les séances de TP nous allons utiliser l'éditeur de textes EMACS, qui, par l'ajout du mode TUAREG permet d'exécuter du code CAML .

Pour lancer EMACS cliquez sur l'icône de EMACS (représentée ci-dessous). Une fenêtre s'ouvre.

Dans le menu FILE choisir VISIT NEW FILE : avec la fenêtre de dialogue qui s'ouvre, choisir votre répertoire de travail et créer le fichier `test.ml`. Il faut impérativement que l'extension du fichier soit `.ml` pour qu'EMACS le voie comme un fichier OCAML .

La fenêtre de EMACS est maintenant celle dans laquelle vous allez taper les phrases CAML . Dans le jargon EMACS on dit que c'est un BUFFER.

Tapez une prase élémentaire CAML , comme `let a = 1;;` par exemple. Demandez à EMACS d'interpréter cette phrase en tapant la combinaison de touches : CTRL-x puis CTRL-e. EMACS vous demande alors dans la ligne de commandes située tout en bas si vous voulez travailler avec OCAML . Comme c'est le cas, tapez juste ENTRÉE pour choisir OCAML .

Désormais vous avez deux fenêtres dans EMACS. Celle du haut est le buffer qui vous permet d'écrire dans le fichier `test.ml`. Celle du bas est la boucle interactive (dite aussi TOPLEVEL) qui présente le résultat des évaluations des phrases.

Vous devriez y lire quelque chose du genre :

```
let a = 1;;

# val a : int = 1
#
```

Si c'est le cas tout va bien !

Attention EMACS n'a pas cédé à la pression du monde windows ou mac, ce qui fait que les raccourcis habituels que vous connaissez, par exemple pour le copier-coller sont différents. Il y a de toutes façons toujours moyen de procéder à la souris en cliquant dans le bon menu.

Voici les principaux raccourcis dont vous aurez besoin dans EMACS.

- Executez la phrase courante (celle où est situé le curseur) : CTRL-x puis CTRL-e (celui là vous le connaissez déjà)
- Exécutez tout le contenu du buffer : CTRL-x puis CTRL-b
- TRÈS IMPORTANT sauvegardez le buffer : CTRL-x puis CTRL-s. Il faudra penser à le faire régulièrement, pour éviter qu'une malheureuse coupure de courant ou un plantage vous fasse perdre tout votre travail.

  Si vous voulez vraiment utiliser les raccourcis claviers pour le copier coller :
- Couper CTRL-w
- Copier ALT-w
- Coller CTRL-y
- Undo CTRL-x puis u (Attention ce n'est pas CTRL-x puis CTRL-u !)

Il peut arriver que vous écriviez du code qui ne s'arrête pas... Il y a moyen d'interrompre l'exécution en passant par le menu TUAREG\INTERACTIVE MODE\INTERRUPT TOPLEVEL.

II Début en douceur

Le but de cette partie est de repasser rapidement en revue ce qui a été présenté en cours pour être sûr de tout bien avoir compris. Il est vivement conseillé d'anticiper la réponse de l'interprèteur puis de vérifier votre supposition.

Dès que quelque chose n'est pas clair après quelques minutes de réflexions, il faut me demander des éclaircissements.

Vous êtes en autonomie pour cette partie.

Pour gagner du temps de frappe (mais pas de réflexion) j'ai pré-tapé les phrases à évaluer dans le fichier `TP1.ml`.

Dès lors téléchargez ce fichier à partir de la plateforme Moodle (ou hugo-prepas). Ouvrez-le en passant par le menu FILE\VISIT NEW FILE : avec la fenêtre de dialogue qui s'ouvre, choisir le répertoire de travail où vous avez téléchargé le fichier `TP1.ml`, le sélectionner et l'ouvrir.

CAML est un langage fonctionnel orienté objet. Tout ce que l'on manipule (constantes, variables, fonctions, etc...) est un objet. Les objets sont distingués par leur nom, leur type et leur valeur.

La commande suivante :

```
# let x = 2 ;;
  x : int = 2
```

définit un objet appelé x, qui a pour valeur 2, et qui a le même type que 2.

Cet objet est donc de type int , c'est-à-dire un entier relatif (integer en anglais). Nous verrons aussi le type float des flottants, qui forment une bonne représentation des réels.

II.1 Activité 1

Tapez les commandes suivantes :

```
1  # 2 + 4;;
2  # let x = 3 + 1;;
3  # -4 * (3 + 2);;
4  # 2 - 1 * 0 + 3;;
5  # 12 / 5;;
6  # 12 mod 5;;
7  # -12 mod 5;;
8  # max_int;;
9  # 3000000000000000000;;
10 # 30000000000000000000;;
11 # 3000000000000000000 * 2;;
```

Les parenthèses indiquent quelle est la priorité à respecter lors des calculs. La plupart du temps, CAML calcule de la gauche vers la droite, mais les priorités entre les opérations $+ - * /$ sont respectées.

Notez la différence de forme entre les réponses selon que l'on évalue une expression ou qu'on définit une liaison globale avec `let ident = expr;;` .

Notez le résultat surprenant de la 7ième phrase : il ne faut pas utiliser `mod` avec des entiers négatifs.

`max_int` rend le plus grand entier positif manipulable dans OCAML . La ligne 8 montre ce qu'il arrive si on écrit un entier plus grand que `max_int` et la ligne 9 ce qu'il risque d'arriver si on calcule un entier trop grand !

II.2 Activité 2

Tapez les commandes suivantes et observez bien les réponses de CAML :

```
1 # 1 + 2.5;;
2 # 1. +. 2.5;;
3 # 4. *. 1.2 -. 0.8;;
4 # 2. **. 10.;;
5 # 3e5 -. 1e1;;
6 # let x = 1.5 in x +. 2.;;
7 # 2. ** 62. -. 1.;;
8 # float_of_int max_int;;
```

CAML se sert des types pour détecter une erreur. Par exemple, on ne peut pas additionner des int et des float .

Il y a par exemple deux opérateurs distincts `+` et `+.` pour l'addition.

La ligne 6 montre qu'on ne met pas de point pour montrer qu'un identifiant correspond à un float .

Les lignes 7 et 8 montrent qu'elle est la valeur de `max_int` sur une machine 64 bits.

Corrigez la deuxième phrase fautive.

II.3 Activité 3

Tapez dans l'ordre les commandes suivantes :

```
1 # 2 * x + 1;;
2 # x = 7;;
3 # let x = 7;;
4 # 2 * x + 1;;
5 # let x = 2 * x + 1;;
6 # let y = x;;
7 # let x = (x - 1) / 2;;
```

Les définitions en CAML peuvent être globales ou locales.

Quand on définit globalement une variable déjà définie, sa valeur précédente est écrasée. On utilise la syntaxe `let ... in ...` pour faire une définition locale.

La ligne 2 n'est pas une affectation ou la définition d'une liaison ! Qu'est ce que c'est ?

II.4 Activité 4

Tapez dans l'ordre les commandes suivantes :

```
1 # let x = 0 ;;
2 # let x = 1 in 10 * x ;;
3 # let x = 1 in let y = 10 * x ;;
4 # let y = let x = 1 in 10 * x ;;
5 # x ;;
```

Quel est le problème de la ligne 3 ?

Par ailleurs il est préférable de ne pas utiliser le même nom pour une liaison locale et pour une liaison globale.

Comme en mathématiques, on peut définir des fonctions en CAML .

II.5 Activité 5

Tapez les commandes suivantes :

```
1 # let f (x : int) : int = 2 * x + 1;;
2 # f 0 ;;
3 # f 10;;
4 # f(x);;
5 # f 0.5 ;;
```

Lors de la définition de la fonction f, x est un paramètre muet, il se comporte donc comme l'identification du liaison locale.

Ici, lorsqu'on appelle `f x` , CAML utilise la définition globale de x.

Attention ! La syntaxe `f x` ne permet pas de remonter à la définition de f.

Et pire, si x est déjà défini, on obtient un résultat aberrant au lieu d'un message d'erreur normal, et on peut ne pas s'en rendre compte !

Une fonction est vue comme un objet pour CAML , elle a donc un type. Dans le cas précédent, f a pour type int -> int .

Le premier int est le domaine de définition de la fontion, c'est-à-dire le type de la variable muette x. Le deuxième int est l'ensemble d'arrivée de la fonction.

Désormais on précisera comme cela a été fait ici le type des paramètres et celui de la valeur calculées par la fonction.

On peut définir les fonctions de plusieurs façons :

```
# let suivant = function (x : int)  -> (x + 1 : int);;
# let suivant (x : int) : int = x + 1;;
```

et les utiliser de plusieurs façons :

```
# suivant 3;;
# let x = 3 in suivant x;;
```

II.6 Activité 6

Tapez les commandes suivantes :

```
# let id x = x;;
# id 3;;
# id 4.5;;
# id (id 3);;
# id id 3;;
```

Lorsqu'on définit une fonction, CAML essaye de deviner son type. Mais ce n'est pas toujours possible. CAML utilise alors des types muets. La notation 'a->'a signifie que la fonction `id` a le même ensemble de départ que d'arrivée et que ceux-ci ne sont pas définis a priori.

La fonction `id` que l'on a défini ici est **polymorphe** !

Expliquer ce qu'il se passe dans le calcul de la ligne 5 (pas facile !).

II.7 Activité 7

CAML possède le type booléen bool qui peut prendre l'une des deux valeurs suivantes : `true` et `false` .

```
# 1 < 2;;
# 1 <= 1;;
# 1 < 2 || 1. > 2.;;
# not 1 = 2 && 1 < 1;;
# 1/0 < 2 && 2 < 1;;
# 2 < 1 && 1/0 < 2;;
```

Pour expliquer la différence de comportement entre les lignes 5 et 6 il faut savoir que les opérateurs booléens utilisent l'évaluation paresseuse.

Dans le cas du et logique `&&` , l'expression de gauche est évaluée. Si elle rend `false` alors OCAML ne cherche même pas à évaluer l'expression de droite car le et logique sera forcément faux.

De même dans le cas du logique `||` , l'expression de gauche est évaluée. Si elle rend `true` alors OCAML ne cherche même pas à évaluer l'expression de droite car le ou logique sera forcément vrai.

Donc ici dans la ligne 6 la division par zéro ne sera provoquée !

Les expressions booléennes nous permettront d'exprimer des conditions dans les gardes des motifs de filtrage, et plus tard avec l'instruction `if ... then ... else ...`

II.8 Activité 8

Tapez les commandes suivantes ;

```
1 # (true, 0);;
2 # 1, 0;;
3 # fst (1, 2);;
4 # snd (1, 2);;
5 # 1, (2, 3);;
6 # 1, 2, 3;;
7 # fst (1, 2, 3);;
```

Notez et expliquez la différence des types rendus pour les lignes 5 et 6.

III À vous de jouer !

Dans cette partie l'enseignant fait le point à chaque question. Et on réfléchit sur le papier avant de se jeter sur la machine.

On précisera comme vu en cours le type des paramètres des fonctions, ainsi que celui de la valeur calculée par la fonction.

III.1

Écrire une fonction `cree_couple` qui prend deux paramètres `x` et `y` de types quelconques et qui calcule le couple `(x, y)` .

Écrire une fonction `swap` renvoyant un nouveau couple dont l'ordre des éléments a été échangé.

Écrire une fonction `first` renvoyant le premier élément d'un couple.

Écrire une fonction `fourth` renvoyant le quatrième élément d'un quadruplet.

Écrire une fonction `map3` de type ('a -> 'b) -> 'a * 'a * 'a -> 'b * 'b * 'b telle que l'appel `map3 f (a, b, c)` renvoie le triplet obtenu en appliquant `f` à chaque composant. Vérifier en testant le code suivant `map3 (function x -> x * x) (1, 2, 3) ;;` .

III.2

Écrire une fonction `quadruple` de type int -> int qui définira une fonction locale `double` de type int->int et qu'elle utilisera pour calculer la valeur égale à quatre fois celle reçue en paramètre.

III.3 Suite de Collatz

La suite de Collatz est définie par $u_0 \in \mathbb{N}$ $u_{n+1} = f(u_n)$ où $f : \mathbb{N} \mapsto \mathbb{N}$ est définie par $f(x) = x/2$ si x est pair et $f(x) = 3x + 1$ si x est impair.

Écrire avec filtrage la fonction f correspondante.

Écrire une fonction récursive `collatz` telle que `collatz u0 n` calcule le n-ième terme de la suite de collatz dont le premier terme est u_0.

III.4

Écrire une fonction `trinome` de type float -> float -> float -> float * float qui prend en paramètre les trois coefficients a, b et c d'un trinôme $ax^2 + bx + c$ et qui rend les deux racines réelles distinctes lorsqu'elles existent et qui lève une exception dans le cas contraire.

III.5

On représente les nombres complexes sous la forme d'un couple de flottants `(partie réelle, partie imaginaire)` (on verra plus tard dans l'année comment on définir un nouveau type).

Écrire les fonctions `re` , `im` , `conjugue` , `modu` , `addition` , `multiplication` et `division` qui calculent respectivement la partie réelle, la partie imaginaire, le complexe conjugué, le module d'un nombre complexe puis la somme, le produit et le rapport de deux nombres complexes.

Pour la fonction `addition` on écrira deux versions pour illustrer deux façons de déconstruire les nombres complexes reçus en paramètre. La définition commencera par `let addition (x1, y1) (x2, y2) = ...` dans le premier cas et par `let addition z1 z2 = ...` dans le deuxième cas.

Pour la fonction `division` on utilisera astucieusement **uniquement** certaines des fonctions précédentes et la définition commencera nécessairement par `let division z1 z2 = ...` (et elle calculera $z1/z2$)

Vérifier que votre fonction `division` rend le bon résultat sur l'exemple suivant :

```
# division (4., 5.) (6., 7.);;
- : float * float = (0.694117647058823506, 0.0235294117647058543)
```

III.6

Écrire une fonction CAML de chacun des types suivants :

1. int -> int
2. int * int -> int
3. int -> int -> int
4. int -> int * int
5. (int -> int) -> int

IV Fontions récursives

IV.1 Fonctions puissance

Vous avez surement noté qu'il n'y a pas d'opération puissance sur les entiers. Nous allons pallier ce manque.

Écrire une première version récursive de la fontion `puissance x n` de type int -> int -> int naïve s'appuyant sur le fait que pour $n > 0$:

$$x^n = x \times x^{n-1}.$$

On montrera que la complexité temporelle d'une telle fonction serait alors linéaire en n (c'est facile à voir...)

Il existe une autre approche s'appuyant sur la méthode dite **diviser pour régner** que l'on étudiera sous peu qui consiste à remarquer que si $n > 0$ est pair alors

$$x^n = (x^2)^{(n/2)}$$

et si n est impair alors

$$x^n = x \times (x^2)^{(n-1)/2}.$$

Cette technique porte le nom **d'exponentiation rapide**. On montrera que sa complexité temporelle est alors logarithmique !

Écrire la fonction réursive `puissance_rapide x n` de type int -> int -> int qui calcule récursivement x^n en utilisant la technique de l'exponentiation rapide.

IV.2

Écrire une fonction récursive `fibo` de type int -> int calculant le n-ième terme de la suite de FIBONACCI définie par $f_0 = 0$, $f_1 = 1$ et $\forall n > 1$: $f_n = f_{n-1} + f_{n-2}$. Pour le moment on se contentera de la traduction naïve de la définition.

Utiliser votre fonction pour calculer f_0, f_{10}, f_{20}, f_{30}, f_{40} (là il faut être un peu patient) et f_{50} (là vous avez le droit de craquer et d'interrompre le calcul...).

On verra plus tard en cours pourquoi la méthode naïve est une très mauvaise méthode.

IV.3 n-ième terme d'une suite récurrente

On donne une suite définie par son premier terme u_0 et la relation de récurrence :

$$\forall n \in \mathbb{N}, u_{n+1} = f(u_n),$$

où f est une fonction de $\mathbb{R}$ dans $\mathbb{R}$.

Écrire une fonction récursive `niemeterme` de paramètres u_0 (flottant), n (entier) et f permettant le calcul du terme d'indice n de la suite. On prédira d'abord le type que CAML attribue à cette fonction.

Utiliser cette fonction en lui passant une fonction anonyme permettant de calculer une approximation de $\sqrt{2}$ selon la méthode dite de Héron qui utilise la limite de la suite définie par : $u_0 = 1$, $u_{n+1} = \frac{1}{2}(u_n + 2/u_n)$.

IV.4 Dérivée p-ième de $x \to x^n$

On considère un monome en x, de coefficient a_n, de puissance n : $a_n x^n$, que l'on représentera par le couple (a_n, n) de type float * int .

Les fonctions dérivées sucessives de la fonction $x \to a_n x^n$ sont également de la même forme et on pourra donc coder le monôme correspondant de la même manière.

Écrire une fonction récursive `coefficient p (a, n)` calculant le couple décrivant le monôme associé à la dérivée p-ième de $x \to ax^n$.